Extending Tuscany

Apache Tuscany

Contents

- What can be extended?
- How to add an extension module?
- How to add an implementation type?
- How to add a binding type?
- How to add a interface type (TBD)
- How to add a data binding type?

What can be extended?

- The SCA assembly model can be extended with support for new interface types, implementation types. and binding types. Tuscany is architected for extensibilities including:
  - Implementation types
  - Binding types
  - Data binding types
  - Interface types

Add an extension module

The Tuscany runtime allows extension modules to be plugged in. Tuscany core and extension modules can also define extension points where extensions can be added.

Life cycle of an extension module

- During bootstrapping, the following sequence will happen:
  - All the module activators will be discovered by the presence of a file named as META-INF/services/org.apache.tuscany.spi.bootstrp.ModuleActivator.
  - The activator class is instantiated using the no-arg constructor.
  - ModuleActivator.getExtensionPoints() is invoked for all modules and the extension points contributed by each module are added to the ExtensionRegistry.
  - ModuleActivator.start(ExtensionRegistry) is invoked for all the modules. The module can then get interested extension points and contribute extensions to them. The contract bwteen the extension and extension point is private to the extension point. The extension point can follow similar patterns such as Registry. If it happens that one extension point has a dependency on another extension point, they can linked at this phase.
- During shutting down, the stop() method is invoked for all the modules to perform cleanups. A module can choose to unregister the extension from theextension points.

Add an extension module

- Implement the org.apache.tuscany.core.ModuleActivator interface. The implementation class must have a no-arg constructor. The same instance will be used to invoke all the methods during different phases of the module activation.
- Create a plain text file named as META-INF/services/org.apache.tuscany.core.ModuleActivator.
- List the implementation class name of the ModuleActivator in the file. One line per class.
- Add the module jar to the classpath (or whatever appropriate for the hosting environment).

The responsibilities of implementation and binding providers

The Tuscany runtime allows extension modules to be plugged in. Tuscany core and extension modules can also define extension points where extensions can be added.

Implementation Provider

- It's contracted by the SPI:

  org.apache.tuscany.sca.provider.ImplementationProvider
- It is responsible to create invokers for components implemented by this type
- It can react to the lifecycle of the components by the start()/stop() callback methods

Reference Binding Provider

- It's contracted by the SPI:

  org.apache.tuscany.sca.provider.ReferenceBindingProvider
- It is responsible to create invokers for outbound invocations over this binding type. The invoker delegates the call to its binding protocol/transport layer.
- It can react to the lifecycle of the components by the start()/stop() callback methods

Service Binding Provider

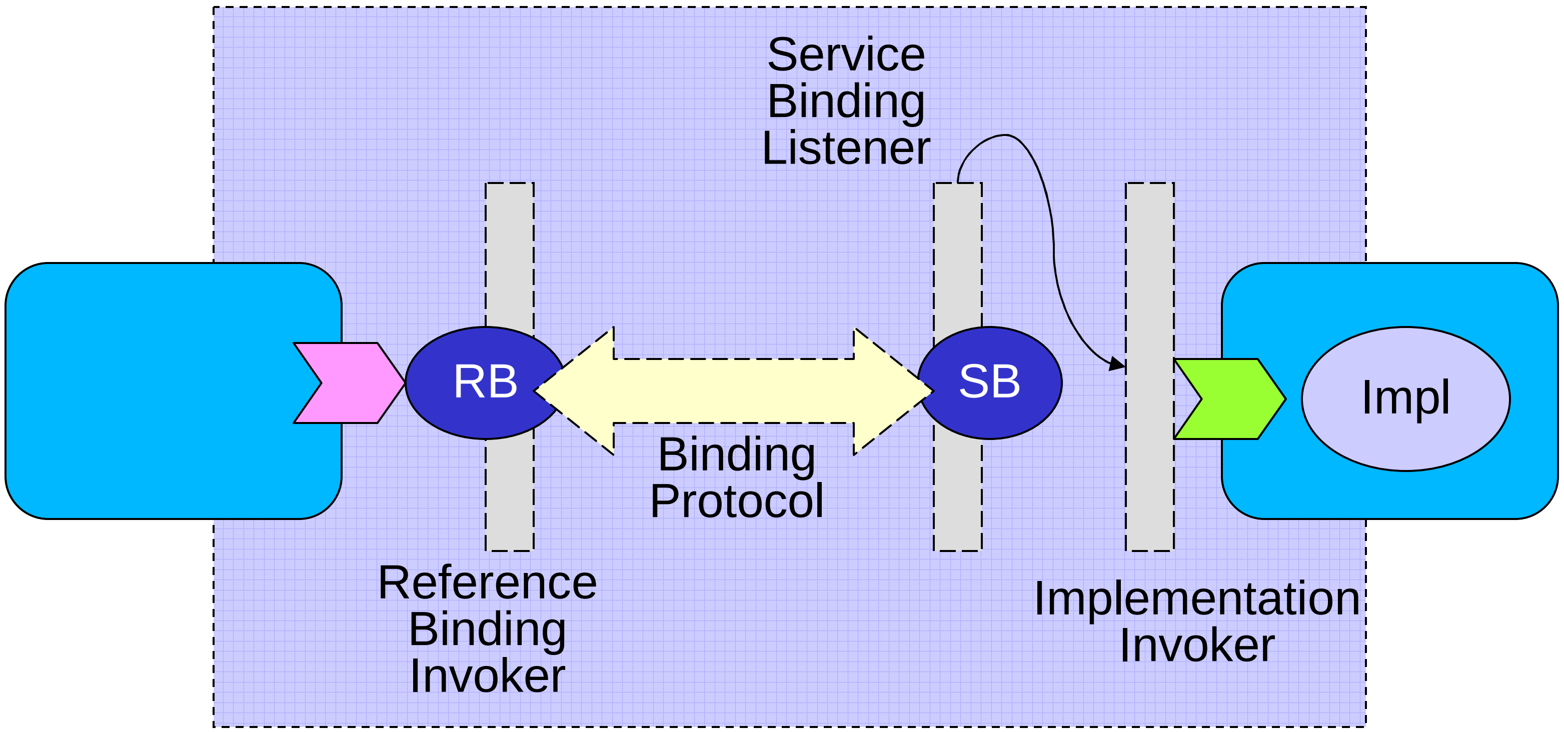
- It's contracted by the SPI:

  org.apache.tuscany.sca.provider.ServiceBindingProvider

- It is responsible to create binding protocol/transport specific listeners for inbound invocations over this binding type. The listener takes the message from the protocol layer and then dispatch it to the promoted component using the invocation chain.

- Usually, it will use the start() method to register the listener and use the stop() method to unregister the listener

The runtime wire and invocation chain



•The reference binding provider contributes the reference binding invoker
•The service binding provider contributes the service binding listener
•The implementation provider contributes the Implementation invoker

Add an implementation type

SCA allows you to choose from any one of a wide range of implementation types, such as Java, Scripting, BPEL or C++, where each type represents a specific implementation technology.

Add a new implementation type

1. Define the assembly model extension
2. Define interfaces/classes to represent the model
3. Implement the StAXArtifactProcessor SPI to read/resolve/write the model
4. Implement the ImplementationProvider SPI to add the invocation/lifecycle control logic
5. Implement the ModuleActivator SPI to hook the StAXArtifactProcessor and ImplementationProvider
6. Contribute an extension module

Extend the assembly model

- The implementation type is referenced by an XML element (implementation.crud) in the composite file

```
<component name=“CRUDComponent”>
        <crud:implementation.crud directory=“/tmp”/ xmlns:crud=“http://crud”>
</component>
```

- The model extension follows the XML inheritance but <u>NO</u> XML schema is required by the runtime

```
 <element name="implementation.crud" type="sca:CRUDImplementation"
       substitutionGroup="sca:implementation" />
   <complexType name="CRUDImplementation">
      <complexContent>
         <extension base="sca:Implementation">
            <attribute name="directory" type="string" use="optional" />
         </extension>
      </complexContent>
   </complexType>
```

Define and process the model

- The model can be simply written as java interfaces and classes and it typically consists of 4 parts
  - The CRUDImplementation interface which extends org.apache.tuscany.sca.assembly.Implementation
  - The CRUDImplementationFactory interface which defines a createImplementation() method
  - The default implementation of CRUDImplementation
  - The default implementation of CRUDImplementationFactory
- Provides an implementation of StAXArtifactProcessor to read/write the model objects from/to XML

Provide the invocation logic

- The logical model can be associated with ImplementationProvider interface to provide invocation logic for the given component implementation type
  - CRUDImplementationProvider implements the ImplementationProvider interface
  - Methods on ImplementationProvider SPI
    - createInterceptor(): Create an interceptor to invoke a component with this implementation type
    - createCallbackInterceptor(): Create a callback interceptor to call back a component with this implementation type

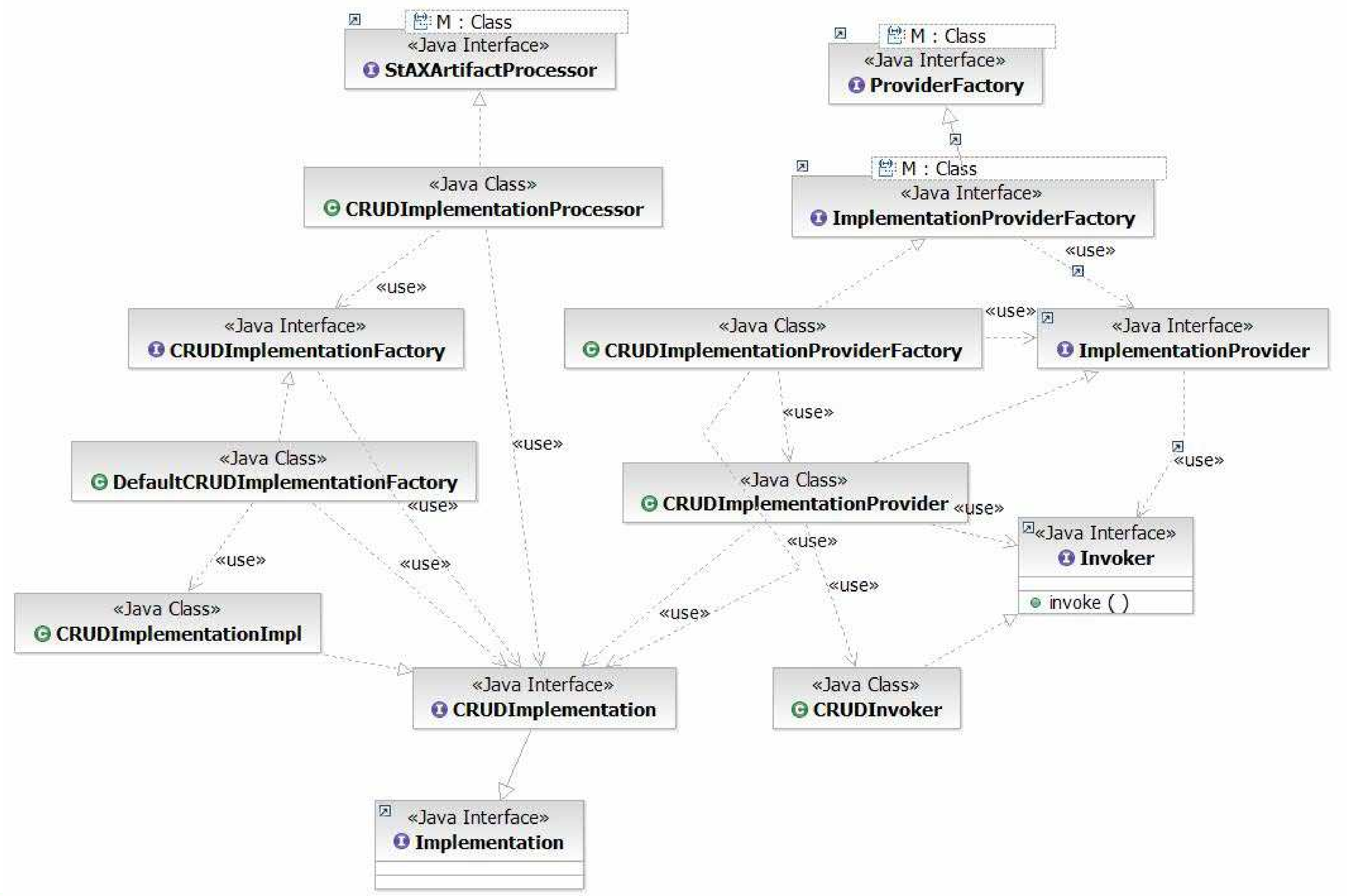
Control the life cycle of components

- The start()/stop() methods
  - start(): A method to be invoked when a component with this implementation type is started. (We simply print a message for the CRUD)
  - stop(): A method to be invoked when a component with this implementation type is stopped. (We simply print a message for the CRUD)

Plug in the implementation type

- The extension module containing the CRUD implementation type can be plugged into Tuscany as follows:
  - Implement the ModuleActivator SPI and register the implementation class in META-INF/services/org.apache.tuscany.sca.core.ModuleActivator
  - Interact with the ExtensionPointRegistry
    - start(): Register the CRUDImplementationProcessor with StAXArtifactProcessorExtensionPoint and Register the CRUDImplementationProviderFactory with the ProviderFactoryExtensionPoint
    - stop(): Unregister the CRUDImplementationProcessor and the CRUDImplementationProviderFactory

The big picture (class diagram)



Add a binding type

References use bindings to describe the access mechanism used to call a service. Services use bindings to describe the access mechanism that clients have to use to call the service.

Add a new binding

1. Define the model extension for reference and service binding
2. Define interfaces/classes to represent the model for the binding
3. Implement the StAXArtifactProcessor to read/resolve/write the models
4. Add the runtime logic by implementing the BindingProviderFactory, ReferenceBindingProvider, ServiceBindingProvider SPIs
5. Implement the ModuleActivator interface to hook up the StAXArtifactProcessor and BindingProviderFactory with respective extension points
6. Contribute an extension module to Tuscany

Extend the assembly model - binding

- The implementation type is referenced by an XML element (binding.echo) in the composite file

```
<reference name="EchoReference" promote="EchoComponent/echoReference">
    <interface.java interface="echo.Echo"/>
    <binding.echo uri="http://tempuri.org" />
  </reference>
```

- The model extension follows the XML inheritance but NO XML schema is required by the runtime

```
<element name="binding.echo" type="sca:EchoBinding"
    substitutionGroup="sca:binding" />
  <complexType name="EchoBinding">
    <complexContent>
        <extension base="sca:Binding"/>
  </complexContent>
  </complexType>
```

Define and process the model

- The model can be simply written as java interfaces and classes and it typically consists of 4 parts
  - The EchoBinding interface which extends org.apache.tuscany.sca.assembly.Binding
  - The EchoBindingFactory interface which defines a createEchoBinding() method
  - The default implementation of EchoBinding (EchoBindingImpl)
  - The default implementation of EchoBindingFactory (DefaultEchoBindingFactory)
- Provides an implementation of StAXArtifactProcessor to read/write the model objects from/to XML
  - EchoBindingProcessor

Provide the outbound invocation logic

- The logical model can be associated with ReferenceBindingProvider interface to provide invocation logic for the given binding type
  - EchoBindingProvider implements the ReferenceBindingProvider interface
  - Methods on ReferenceBindingProvider SPI
    - createInterceptor(): Create an interceptor to invoke a component with this binding type
    - createCallbackInterceptor(): Create a callback interceptor to call back a component with this binding type
    - getBindingInterfaceContract(): Get the interface contract imposed by the binding protocol layer
    - start(): Lifecycle callback method that can be used by the reference binding to allocate resources or create connections
    - stop(): Lifecycle callback method that can be used by the reference binding to do some house keeping.

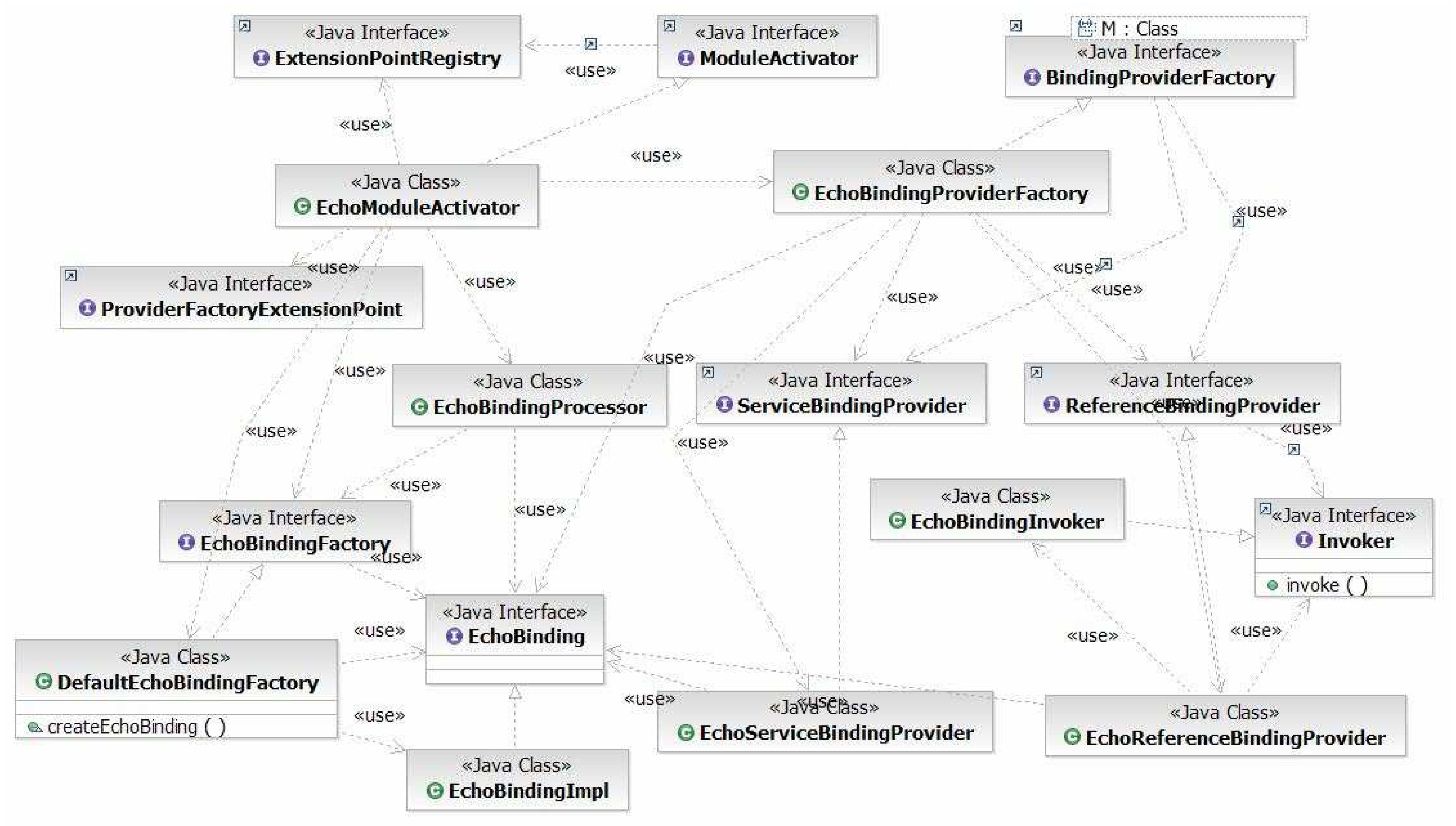
Provide the inbound invocation logic

- The logical model can be assoicated with ServiceBindingProvider interface to provide invocation logic for the given binding type
  - EchoBindingProvider implements the ServiceBindingProvider interface
  - Methods on ServiceBindingProvider SPI
    - start(): Start the binding-specific protocol listener to receive incoming messages from the transport layer. The listener will be responsible to dispatch the call to the promoted component.
    - stop(): Stop the binding-specific protocol listener
    - getBindingInterfaceContract(): Get the interface contract imposed by the binding protocol layer

Plug the binding type into Tuscany

- The extension module containing the ECHO binding type can be plugged into Tuscany as follows:
  - Implement the ModuleActivator SPI and register the implementation class in META-INF/services/org.apache.tuscany.sca.core.ModuleActivator
  - Interact with the ExtensionPointRegistry
    - start(): Register the EchoBindingProcessor with StAXArtifactProcessorExtensionPoint and EchoBindingProviderFactory with ProviderFactoryExtensionPoint
    - stop(): Unregister the EchoBindingProcessor/EchoBindingProviderFactory

The big picture (class diagram)


«Java Interface» ExtensionPointRegistry
«Java Interface» ModuleActivator
M : Class
«Java Interface» BindingProviderFactory
«Java Class» EchoModuleActivator
«Java Class» EchoBindingProviderFactory
«Java Interface» ProviderFactoryExtensionPoint
«Java Class» EchoBindingProcessor
«Java Interface» ServiceBindingProvider
«Java Interface» ReferenceBindingProvider
«Java Class» EchoBindingInvoker
«Java Interface» Invoker
invoke ( )
«Java Interface» EchoBindingFactory
«Java Interface» EchoBinding
«Java Class» DefaultEchoBindingFactory
createEchoBinding ( )
«Java Class» EchoServiceBindingProvider
«Java Class» EchoReferenceBindingProvider
«Java Class» EchoBindingImpl
«use»

Add a data binding

Tuscany provides a data binding framework to allow business data to be represented in the preferred way chosen by the components. New data bindings and transformers can be contributed to Tuscany to facilitate the data transformations.

Add a new databinding

- The data binding is identified by a string id. A data binding can also have aliases.
- Adding a databinding can be as simple as just adding a transformer that references the databinding id. For example, if a transformer that transforms data from “db1” to “db2” is added, then “db1” and “db2” is active.

Add a new databinding

- If the databinding needs to deal with advanced capabilities, such as:
    - Introspect the data types to recognize data of this binding
    - Handle operation wrapping/unwrapping
    - Handle copy of data
    - Handle exception transformations
- Then, you need to implement the o.a.t.s.databinding.DataBinding SPI

Add a new transformer

- A transformer is responsible to transform data from one databinding to another one
- Adding a transformer to Tuscany will enrich the transformation capabilities as it adds more links to the transformation graph.
- Provide a transformer is simply to implement the PullTransformer/PushTransformer interface.

Sample Transformer Code

```
public class OMElement2XMLStreamReader extends BaseTransformer<OMElement, XMLStreamReader>
      implements PullTransformer<OMElement, XMLStreamReader> {

      public XMLStreamReader transform(OMElement source, TransformationContext context) {

        return source.getXMLStreamReader();

    }

    public Class getSourceType() {

        return OMElement.class;

    }

    public Class getTargetType() {

        return XMLStreamReader.class;

    }

    public int getWeight() {

        return 10;

    }

}
```

Register the data binding/transformer

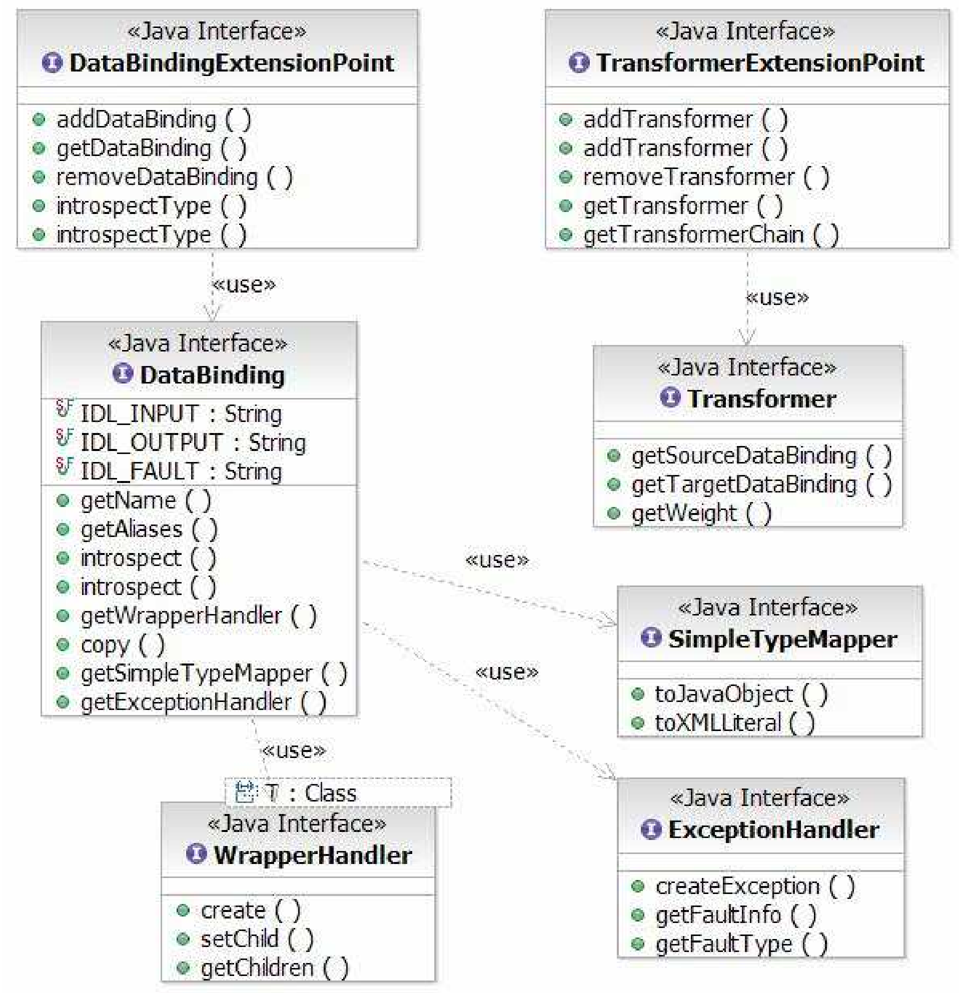
- The databindings and transformers can be registered against the DataBindingExtensionPoint and TransformerExtensionPoint in the ModuleActivator.start() method.

```
public void start(ExtensionPointRegistry registry) {
        DataBindingExtensionPoint dataBindings = registry.getExtensionPoint(DataBindingExtensionPoint.class);
        dataBindings.addDataBinding(new AxiomDataBinding());
        TransformerExtensionPoint transformers = registry.getExtensionPoint(TransformerExtensionPoint.class);
        transformers.addTransformer(new Object2OMElement());
        transformers.addTransformer(new OMElement2Object());
        transformers.addTransformer(new OMElement2String());
        transformers.addTransformer(new OMElement2XMLStreamReader());
        transformers.addTransformer(new String2OMElement());
        transformers.addTransformer(new XMLStreamReader2OMElement());
    }
```

Plug the binding type into Tuscany

- The extension module containing the data binding type can be plugged into Tuscany as follows:
  - Implement the ModuleActivator SPI and register the implementation class in META-INF/services/org.apache.tuscany.sca.core.ModuleActivator
  - Interact with the ExtensionPointRegistry
    - start(): Register the databinding with DataBindingExtensionPoint and transformers with TransformerExtensionPoint
    - stop(): Unregister the databinding from DataBindingExtensionPoint and transformers from TransformerExtensionPoint

The big picture (class diagram)


«Java Interface»
DataBindingExtensionPoint
addDataBinding ( )
getDataBinding ( )
removeDataBinding ( )
introspectType ( )
introspectType ( )
«Java Interface»
TransformerExtensionPoint
addTransformer ( )
addTransformer ( )
removeTransformer ( )
getTransformer ( )
getTransformerChain ( )
«use»
«use»
«Java Interface»
DataBinding
IDL_INPUT : String
IDL_OUTPUT : String
IDL_FAULT : String
getName ( )
getAliases ( )
introspect ( )
introspect ( )
getWrapperHandler ( )
copy ( )
getSimpleTypeMapper ( )
getExceptionHandler ( )
«Java Interface»
Transformer
getSourceDataBinding ( )
getTargetDataBinding ( )
getWeight ( )
«use»
«Java Interface»
SimpleTypeMapper
toJavaObject ( )
toXMLLiteral ( )
«use»
«use»
T : Class
«Java Interface»
WrapperHandler
create ( )
setChild ( )
getChildren ( )
«Java Interface»
ExceptionHandler
createException ( )
getFaultInfo ( )
getFaultType ( )